SCIS 2005 The 2005 Symposium on Cryptography and Information Security
Maiko Kobe, Japan, Jan.25-28, 2005
The Institute of Electronics, Information and Communication Engineers

Construction of RBAC-Enforceable Security Automata

Hyung Chan Kim * Wook Shin † R. S. Ramakrishna ‡ Kouichi Sakurai §

Abstract— Conventional access control models have been supported confidentiality and integrity reflecting required organizational security policy. However, attacks involving operational semantics or concurrency can not be considered with generic access control because of its behavioral characteristic. This paper propose an RBAC-Enforceable Security Automata to trace the sequential access events of trusted operating systems hardened with Role-Based Access Control (RBAC). Product construction of such an automaton can detect the condition of time-of-check-to-time-of-use (TOCTTOU) attack which concerns with concurrency.

Keywords: access control, behavior control, time-of-check-to-time-of-use, role-based access control, operating system security, trusted operating systems, automata

1 Introduction

We have been widely adopted role-based access control (RBAC) [1, 2] as major access control enforcement mechanism in variable systems including operating systems, since it offers flexibility in enforcement, and policy neutrality. It also lessens the burden of administration. Trusted Operating Systems (TOS), an operating system which includes a security kernel providing protection from diverse unknown threats, introduce RBAC as their core access control mechanism in many research projects as well as commercial products.

TOS involves operational dynamics: each processes produce a sequence of operational semantics, and sometimes a set of processes interoperate via inter-process communication (IPC) facilities such as file, socket, shared memory, and so on. However, the prevalent research trend of RBAC has intensively illuminated the reflection of organizational security policy, not the behavior of overall system executions. In other words, researches have focused on the engineering of formation amongst access entities by making relations between subject and object based on their security criterion. With this enforcement, operations are permitted under the policy approval but the order of operations is not restricted.

There is a class of attack in that the order of operations is important. Attacks exploiting the runtime program environment of setuid (set user id) assigned application in UNIX operating system is one of such example. In terms of access control, setuid mechanism can be considered as temporal privilege transition to the other privilege. Sending a mail or changing password requires such kind of operations. Many runtime program attacks – such as buffer overflow, format string, double free, and so on – change the execution flow of process under the legal access context: in view of discretionary access control (DAC) in UNIX, such privilege transition via setuid mechanism is permitted, but there is no restriction on the execution sequences. Setting aside DAC-enforced system, any operating system which supports privilege transition in their access control mechanism can be harmed with similar manner [3]. Moreover, attacks involving concurrency of processes, such as time-of-check-to-time-of-use (TOCTTOU) attacks [4], also can not be considered effectively in conventional schemes.

This work focuses on the operational semantics in terms of access control. Performing an atomic operation corresponds to a tuple of subject, object, and operation with the valid session under RBAC policy configuration. A set of tuples can be thought as executions with which we can further enforce behavioral restriction against attacks which exploits the order of operations or concurrency. We propose the construction of security automata for enforcing Role-Based Access Control (RBAC) for Trusted Operating Systems. We model our automata to be EM (Execution Monitor) enforceable [5] so that can restrict the sequential executions of program behavior in operating systems. Moreover, the product construction of execution monitor also shown here to protect TOCTTOU attacks considering the operational concurrency.

The rest of this paper is organized as follows. In section 2, the concept of policy with EM property and RBAC are presented. Section 3 shows how attacks can exploit in view of operations under the legal access context. We propose security automata which have RBAC access context and also make product construction against TOCTTOU attacks in section 4. Discussion will be given in section 5 and the paper ends with conclusions in section 6.

* Department of Information and Communications, Gwangju Institute of Science and Technology (GIST), Gwangju 500-712, Rep. of Korea (kimhc@gist.ac.kr)
† GIST (sunihill@gist.ac.kr)
‡ GIST (rsr@gist.ac.kr)
§ Faculty of Computer Science and Communication Engineering, Kyushu University, Fukuoka 812-8581, Japan (sakurai@csce.kyushu-u.ac.jp)

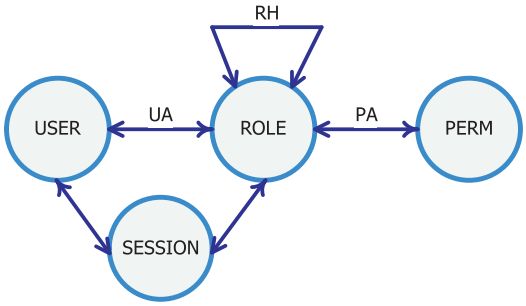

Figure 1: Model of Role Based Access Control (RBAC96)

2 Background

In this section we briefly shows the concept of RBAC, and the concept of Execution Monitor is given based on the work of Alpern, Schnneider, and Bauer [5, 6, 7].

2.1 Role-Based Access Control

The main characteristic of RBAC [1, 2] [Fig. 1] is that it does not directly associate a subject with an object. Instead, by conceiving the *role* which represents job functions or responsibilities in a system or an organization, RBAC greatly eases access control administration. A conventional DAC or a MAC system usually involves direct association between a subject and an object. If there are hundreds of thousands of access entities – a possibility in large enterprises – administrators of DAC or MAC system have difficulty in managing all the access entities. A specific role gathers a set of necessary permissions – defined as the cartesian product of the set of operations and the set of objects – in order to perform a certain duty. Hence, if an administrator of the RBAC system wants to make a subject perform a given duty, then the subject is simply assigned an appropriate role.

The abstraction of role offers several advantages as it enables us to co-opt many useful methods from the field of software engineering. Due to the similarity of roles and class objects, one can adopt object oriented approach just as for class objects. For example, if a role is once codified, then reusability amounts to reassigning subjects to the role of the same responsibility. Similar duties can be easily constructed by modifying only a few attributes of an existing role.

Nowadays, many projects which aim to harden operating system include RBAC: Security-Enhanced Linux [8], GRSecurity [9], and many other commecial TOSs.

2.2 Excution Monitor

2.2.1 EM-Enforceable Security Policy

Schnneider developed the properties of execution which is enforceable under security policy with the concept of execution monitor (EM) [5]. Here we give notions of EM-enforceable systems and security automata. In this work we emphasize operational semantics with access control context.

A system performs a sequence of atomic operations $a_1, a_2, ..., a_n \in \Sigma$, where Σ is a set of operations (access events). We use σ and τ to denote a finte sequence of operations. A system execution Π consists of each atomic operations, therefore $\Pi \subseteq \Sigma^*$ and $\sigma, \tau \in \Pi$. For any sequence $\sigma = a_1 a_2 a_3 ... a_n$,

$$\sigma[i] = a_i$$

$$\sigma[..i] = a_0 a_1 ... a_i$$

$$\sigma[i..] = a_i a_{i+1} ... a_n.$$

A *security policy*, in sense of operational semantics [1] is defined as a predicate on sets of executions. A set of executions Π statisfies a policy P if and only if $P(\Sigma)$ equals true.

A *property* is a set of infinite sequence of program states [6]. A security policy P is intended behavior under execution monitoring if P is specified by a predicate of the form

$$P(\Pi) : (\forall \sigma \in \Pi : \hat{P}(\sigma)) \tag{1}$$

where $\hat{P}$ is a predicate on executions and it decides whether a given individual execution have to be terminated or not.

The condition for a property to be EM-Enforceable is that P must be prefix-closed:

$$(\forall prefix(\sigma), \sigma \in \Pi : \neg \hat{P}(prefix(\sigma)) \implies (\forall \sigma \in \Pi : \neg \hat{P}(\sigma))) \tag{2}$$

Together with (1) and (2), *safety properties* is defined to specify that "bad things" do not happen during execution [6]. If an operational sequence deviates from the criterion, then it must be terminated after some finite operations.

$$(\forall \sigma \in \Pi : \neg \hat{P}(\sigma) \implies (\exists i : \neg \hat{P}(\sigma[..i]))) \tag{3}$$

2.2.2 Security Automata

A safety properties can be recognized by a variant of Büchi Automata: *Security Automata* (SA) is a quadruple.

- Σ: a countable set of access events,
- Q: a countable set of automaton states,
- $q_0 \in Q$: an initial state,
- $\delta : Q \times \Sigma \longrightarrow Q$ is a partial transition function.

Access events are gorverned by SA if a transition is defined during the system execution. Recognition of a sequence of access events means that the sequence is under the currently enforecd policy thereby restricting the behavior of operations. Note that an access event is simply a tuple of object, operation, and subject which is activated context by conventional access control schemes.

[1] We distinguish the term security policy with that of rules that have been unambiguously expressed when enforcing organizational policy such as DAC, MAC, or RBAC.

Table 1: Example of Operational Access Context in UNIX

	Intended Behavior	Exploited Behavior
1	(jim,jim,root,*,*)	(jim,jim,root,*,*)
2	(jim,jim,root,seteuid(root),ϵ)	(jim,jim,root,seteuid(root), ϵ)
3	(jim,root,root,open,/abc)	(jim,root,root,open,/abc)
4	(jim,root,root,read,/abc)	(jim,root,root,read,/abc)
5	(jim,root,root,lseek,/abc)	(jim,root,root,lseek,/abc)
6	(jim,root,root,write,/abc)	(jim,root,root,execve,/bin/sh)
7	(jim,root,root,close,/abc)	(jim,root,root,*,*)
8	(jim,root,root,exit,ϵ)	(jim,root,root,*,*)

3 Scenario of Attacks with Operational Access Context

In this section we analyse two types of attacks which can be viewed with operational semantics in operating systems.

3.1 Attack under Legitimate Access Context

Todays we encounter a various type of program runtime attacks such as Stack Overflow [10], Heap Overflow, Double Free, Format String, and so on. Most attack aims to acquire privileged shell in UNIX/Linux operating systems. The problem is on that conventional access control in UNIX can not effectivly protect the above attack. Precisely speaking, the exploiting process can be achieved under legitimate access control context.

In generic Linux operating system, access control is achived based on permission bits and this is categorized in discretionary access control (DAC). sometimes a user temporailiy needs to transit its privilge for some special purpose. Representative example is chaging password of its own, and during the modification of password file the user proccess is in privileged mode. In UNIX OS, this function is supported by setuid (set user identity) mechanism. User process can be transited its privilege by setting effective uid (euid), to be equal to suid. Therefore, if suid is the uid of super user (zero), user process can perform privileged operation during the transited execution. Finishing prvileged jobs, the privileged mode is terminated by terminating the process: users can execute only defined operations designated by suid program.

However, if a suid program have a runtime bug and an attacker can spawn a shell during the execution, the transited privilge can not be returned and the user process stays at privileged mode beyond its original privilege. For the description of an example, we define access context in UNIX as $\Sigma = (uid, euid, suid, op, obj)$, which is a tuple of uid (subject), effective uid, saved uid, operation, and object, respectively.

Table 1 shows the operational access contexts of a simeple example. Left column contains a sequence of intended behavior and unwanted case is in right column. * means some arbitrary symbol for each item and ϵ means empty. In line 2, the user program transited to its privilege to root (superuser) to write some contents into a protected file (/abc) for special purpose. Note that this is *legal* under the system policy configured by administrator. However, an attacker exploits the program between line 5 and 6 due to some buffer management bug contained in the program and executes a shell. From that point, the attacker can flourish full system privileges and this is still legitimate under the access control enforcement because an attacker only changes its operations and doesn't violate the DAC mechanism: one just prolonged one's staying at privileged level and there are no restriction on the operational semantics.

This kind of attack can be possible in every operating system if the given enforcement mechanism support some sort of privilege transition [3]. We can define $\Sigma = (user, role, opr, obj)$ to depict the behavior of RBAC context. There might be privilege transition by role transition of a user. In case of SELinux [8], it adopt type transition to enable privilege transitions (Role transition was also availble in SELinux and now it is deprecated.) and it is part of type enforcement: another access control mechanism beyond conventional permission based scheme.

3.2 TOCTTOU Attack

Another attack senario is attack involving concurrency known as time-of-check-to-time-of-use (TOCTTOU) attack [4, 11]. TOCTTOU attack can be caused by invalid assumption of programmer that the reference of the resource will not chagne between the time of check and the time of use. Unfortunately, this is not true because modern operating systems handle multiple process concurrently and process scheduler takes charge of its management. We take an archetypal TOCTTOU binding flaw example [4] to see the procdure of the attack in Table 2. The open system call is invoked right after the access system call and programmer do not imagine that something can happen between two consecutive system calls. However, system scheduler is involved and there is possibility of performing another actions by another processes between them. As an example shown here, if the reference of temporary file is modified by unlink and symlink in stage 2 and 3 during the time scheduled for the attacker's program, the system program writes some parameter-offered contents not to the orignal file but to /etc/passwd file in

Table 2: An Archetypal TOCTTOU Attack

	System Program	Attacker's Program
1	(system,root,root,access,/tmp/abc)	
2		(jim,jim,jim,unlink,/tmp/abc)
3		(jim,jim,jim,symlink,(/etc/passwd,/tmp/abc))
4	(system,root,root,open,/tmp/abc)	
5	(system,root,root,write,/tmp/abc)	

stage 5. Hence attacker can insert dummy accounts to /etc/passwd file without having privilege of administrator.

TOCTTOU attacks can be protected if attack signature is obtained from analysis of specific attack case. However, attacks are still vaild even recently [12], and range of attacks are not confied to file race condition but various senario is possible including database systems, or Java class loader [11], thus we have to estabilsh fundamental countermeasure to cope with *unknown TOCTTOU attacks.*

4 A Proposal of Operational Constraint with RBAC-Enforceable Security Automata

In this section, we propose an security automata which can constrain operations in trusted operating systems hardend with RBAC.

4.1 RBAC-Enforceable Security Automata

Our framework simply extend existing RBAC configuration which constrains orgarnizational information flow in a system. Upon RBAC configuration, we introduce Execution Monitors (EM) to track the behavior of operations in terms of RBAC access contexts. Each RBAC contexts represent granted access by the RBAC reference monitor. Even though a given access events is granted to execute by the reference monitor, it is needed to be examined by EM relating with other access events. The behavioral policy can be configured with the language acceptable by the state machine.

The main components of the core RBAC are given below.

- $USER, ROLE, OPR, OBJ$: the set of users, roles, operations and objects.
- $PERM = OPR \times OBJ$: the set of permissions.
- $SESSION$: the set of sessions.
- $UA \subseteq USER \times ROLE$: a many-to-many user-to-role relation.
- $PA \subseteq PERM \times ROLE$: a many-to-many role-to-permission assignment relation.
- $RH \subseteq ROLE \times ROLE$: partial ordering on $ROLE$ called the inheritance relation, i.e.,
  - For $\forall r_1, r_2 \in ROLE_i, r_1 \geq r_2$ means r_1 is an ancestor of r_2. (Equivalently, r_2 is a decendent of r_1.)

Some functions for RBAC relations are defined as follow.

- $users(r : ROLE) \rightarrow 2^{USER}$: users assigned to a role r, namely,
  - $users(r) = \{u \in USER \mid (u, r) \in UA\}$
- $active\_user(s : SESSION) \rightarrow USER$: the mapping from a session to a user.
- $active\_roles(s : SESSION) \rightarrow 2^{ROLE}$: the mapping from a session to a set of roles, i.e.;
  - $active\_roles(s) \subseteq \{r \in ROLE \mid (active\_user(s), r) \in UA\}$
- $perms(r : ROLE) \rightarrow 2^{PERM}$: permissions assigned to a role r, namely,
  - $perms(r) = \{p \in PERM \mid (p, r) \in PA\}$
- $inherited\_perms(r : ROLE) \rightarrow 2^{PERM}$, inherited permissions on role r, namely,
  - $inherited\_perms(r) = \{p \in PERM \mid (\exists r' < r)[(p, r') \in PA]\}$
- $authorized\_perms(r : ROLE) \rightarrow 2^{PERM}$, authorized permissions on role r, defined as,
  - $authorized\_perms(r) = perms(r) \cup inherited\_perms(r)$

RBAC-Enforceable Security Automata (RBAC-SA) is a quintuple: $\langle RBAC, \Sigma, Q, q_0, \delta \rangle$ such that

- $RBAC$: an access control configuration,
- $\Sigma = USER \times ROLE \times OPR \times OBJ$: an access context, i.e.,
  - $\{(user, role, opr, obj) \mid \exists session : SESSION,$ $[user \in users(role) \wedge opr \times obj \in perms(role) \wedge$ $user \in active\_user(role) \wedge$ $role \in active\_role(session) \wedge$ $opr \times obj \in authorized\_perms(role)]\}$
- Q: a countable set of automaton states,
- $q_0 \in Q$: an initial state,

- $\delta : Q \times \Sigma \longrightarrow Q$ is a partial transition function.

With RBAC-SA, $q_0 \xrightarrow{w} q$ for some $q \in Q$ means the acceptance of access events sequence w. The recognition of behavioral policy P by an acceptor M is:

$$\{w \in \Sigma^* \mid \exists q \in Q : q_0 \xrightarrow{w} q\} \tag{4}$$

Therefore, if a given sequence is out of in P, then the behavior is not intended. We can consider the configuration of negative behavior as well as intended behavior with negating the accetable language. With the negative specification, we can constraint the process behavior $seteuid.*.execve$ not to be executed in that sequence against the exemplified attack in Table 1.

Here we defined security automata to be deterministic. However, the choice can be nondeterministic depending on the behavior configuration method.

4.2 Product Construction of RBAC-SA

TOCTTOU attacks involve concurrency among processes and the product construction of states of SA is needed to capture those conditions. Our consideration is on multiple Execution Monitors (EM) which trace each processes in TOS.

For example, thinking with the example in Table 2, an EM (EM_1) which is monitoring the system program and another EM (EM_2) monitoring attacker's process are respectively running in a system. Noting only operations, EM_1 will accept a string $access.open.write$ and EM_2 will accept a string $unlink.symlink$ in sometimes. The idea of detecting TOCTTOU attack is on the merge production of languages of multiple acceptors corresponding each EMs.

- $EM_1 = \langle RBAC, \Sigma, Q_1, 0_1, \delta_1 \rangle$,
- $EM_2 = \langle RBAC, \Sigma, Q_2, 0_2, \delta_2 \rangle$,
- $EM_{||} = \langle RBAC, \Sigma, Q_{||}, 0_{||}, \delta_{||} \rangle$,
  - $Q_3 = Q_1 \times Q_2 = \{(p, q) \mid p \in Q_1 \wedge q \in Q_2\}$
  - $\delta_{||}((q_1, q_2), a) = \{(\delta_1(q_1, a), q_2), (q_1, \delta_2(q_2, a))\}$

$EM_{||}$ traces the evolution of global product states of processes. If $EM_{||}$ detect an instance of merged string such as $access.open.unlink.symlink.write$, as a negative behavior then we can enforce all involving processes have to be terminated as we $EM_{||}$ has decided that the condition of TOCTTOU attack are in current global state. We define a new paradigm of policy mitigating a sort of attacks with concurrency as follow:

- **Conflict of Behavior (Policy) :** $COB(q_1, q_2)$ if two COB states are coincidently met, then involving processes are forced to be terminated.

Pairwise extension of the merge production of SA can enforce on overall processes in a system.

5 Discussion

The consideration of behavioral aspects is mostly the area of intrusion detection. Intrusion Detection Systems (IDS) with the normal behavior database tracks the sequence of processes and terminates a process if it deviates from the database. In view of access control, each atomic operation, a subject operates on an object, passed from IDS sensor is granted with the behavioral policy. Our framework can be thought, in part, as an intended behavior specification based intrusion detection [13]. Because we can built intended behavior specification as acceptable strings by Securiy Automata. However, specification based IDS does not care about the any properties except subject of access control: if we say an access control with specification based inrusion detection, only identity based access control (IBAC) is possible in snapshot of operations, because each behavior specification is identified by corresponding user id.

Our final intention is the construction of unified model of access control with intended behavior constraint in operating systems. Practically, coexistance of enhanced access control and intrusion detection system is very heavy in general operating systems. Moreover, the configuration of trusted systems (with enhanced access control) and intrusion detection systems are not negligible. Security administrator has to attention on both configurations.

If we can support unified security framework comprising both technologies, we can mitigate the budden of security configuration as well as the system performance penalty.

Another issue with simultaneous attention of access control and behavior recognition is *noninterference*. A system is noninterference secure if for all users, their output sequence is the same as the output sequence purged of inputs from higher users. Ko et al. [14] present an intrusion detection based on the concept of noninterference for detecting race condition attacks. We believe that our framework can support this concept with operation traces and role hierarchy.

6 Conclusion

In this paper, we have examined attacks which can not be constrained in static configuration of access control effectively. Considering operational semantics, we have proposed a framework to trace of prosses behavior in operating systems with Security Automata which is EM-Enforceable and its context reflects RBAC access events. Moreover, the merge type of product construction of SA supports the detection of TOCTTOU attacks with conflict of behavior policy enforcement.

We have plans to investigate the applicability of some conventional security policies with EM monitor.

Acknowlegement This research was supported in part by Joint Forum for Strategic Software Research (SSR) of International Information Science Foundation

of Japan, and in part by KAIST/GIST BK21 of Ministry of Education of Korea.

References

[1] D. Ferraiolo, J. Cugini, and R. Kuhn, "Role Based Access Control: Features and Motivations," In Proc. of Computer Security Applications Conference, IEEE Computer Society Press, 1995.

[2] R. S. Sandhu, E. J. Coyne, H. L. Feinstein and R. Chandramouli, "Role-Based Access Control Models," IEEE Computers, Vol. 29, No. 2, pp. 38-47, Feb. 1996.

[3] H. C. Kim, R. S. Ramakrishna, K. Sakurai, "On the Privilege Transitional Attack in Secure Operating Systems," In Proc. of Computer Security Symposium 2004 (CSS2004), Vol. II, pp. 559-564, 2004.

[4] M. Bishop and M. Dilger, "Checking for Race Conditions in File accesses," Computing Systems, Vol. 9, No. 2, pp. 131-152, 1996.

[5] F. B. Schneider, "Enforceable Security Policies," ACM Trans. on Information and System Security, Vol. 3, No. 1, pp. 30-50, Feb. 2000.

[6] B. Alpern and F. B. Schneider, "Recognizing Safety and Liveness," Distributed Computing, Vol. 2, pp. 117-126, Feb. 1987.

[7] L. Bauer, J. Ligatti, and D. Walker, "More Enforceable Security Policies," Tech Report TR-649-02, Princeton Univ., 2002.

[8] P. Loscocco, and S. Smalley, "Integrating Flexible Support for Security Policies into the Linux Operating System," In Proc. of the FREENIX Track: 2001 USENIX Annual Technical Conference (FREENIX'01), 2001. (http://www.nsa.gov/selinux/index.html)

[9] http://www.grsecurity.net/

[10] Alphe One, "Smashing The Stack For Fun And Profit," Phrack Vol.7 Issue. 49, File 14 of 16, 1996.

[11] J. C. Lowery, "A Tour of TOCTTOUs," SANS GSEC practical v.1.4b, Aug 2002.

[12] SecuriTeam, "Wget Race Condition Vulnerability Allows a Symlink Attack," May 2004. (http://www.securiteam.com/)

[13] P. Uppuluri and R. Sekar, "Experiences with Specification-based Intrusion Detection," In Proc. of Recent Advances in Intrusion Detection, LNCS 2212, pp. 172-190, 2001.

[14] C. Ko, and T. Redmond, "Noninterference and Intrusion Detection," In Proc. of IEEE Symposium on Security and Privacy, pp. 177-187, 2002.